(2)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) **Ada Compiler Validation Summary Report:** SYSTEAM KG SYSTEAM Ada Compiler S.7000/BS2000, Siemens 7.530-B (host & target), 890831I1.10188	5. TYPE OF REPORT & PERIOD COVERED 31 Aug 89 - 1 Dec 90 / 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) IABG, Ottobrunn, Federal Republic of Germany.	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS IABG, Ottobrunn, Federal Republic of Germany.	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS **Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081**	12. REPORT DATE / 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) IABG, Ottobrunn, Federal Republic of Germany.	15. SECURITY CLASS (of this report) UNCLASSIFIED / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

SYSTEAM KG, SYSTEAM Ada Compiler S.7000/BS2000, Version 1.81, IABG, West Germany, Siemens 7.530-B under BS2000, Version 7.5A (host & target), ACVC 1.10

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AVF Control Number: IABG-VSR-044

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #890831I1.10188
SYSTEAM KG
SYSTEAM Ada Compiler S.7000/BS2000
Siemens 7.530-B Host and Target

Completion of On-Site Testing:
31th August 1989

Prepared By:
IABG mbH, Abt SZT
Einsteinstr 20
D8012 Ottobrunn
West Germany

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Ada Compiler Validation Summary Report:

Compiler Name: SYSTEAM Ada Compiler S.7000/BS2000
Version 1.81

Certificate Number: #890831I1.10188

Host and Target: Siemens 7.530-B under BS2000, Version 7.5A

Testing Completed Thursday 31th August 1989 Using ACVC 1.10

This report has been reviewed and is approved.

IABG mbH, Abt SZT
Dr S. Heilbrunner
Einsteinstr 20
D8012 Ottobrunn
West Germany

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr John Solomond
Director
Department of Defense
Washington DC 20301

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by &CONTRACTOR& under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed Friday 25th August 1989 at SYSTEAM KG, Karlsruhe.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from

IABG mbH, Abt. SZT
Einsteinstr 20
D8012 Ottobrunn

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.

Inapplicable test — An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.

Passed test — An ACVC test for which a compiler generates the expected result.

Target — The computer which executes the code generated by the compiler.

Test — A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.

Withdrawn test — An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and

tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: SYSTEAM Ada Compiler S.7000/BS2000, Version 1.81

ACVC Version: 1.10

Certificate Number: #890831I1.10188

Host and Target Computer:

Machine :	Siemens 7.530-B
Operating System :	BS2000 Version 7.5A
Memory Size :	6 MB

Communications Network: FITSIE

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

1) This implementation supports the additional predefined types SHORT_INTEGER and SHORT_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

4) No exception is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

5) No exception is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

6) Underflow is not gradual. (See tests C45524A..Z (26 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding the test results indicate the following:

1) The method used for rounding to integer is round away from zero. (See tests C46012A..Z (26 tests).)

2) The method used for rounding to longest integer is round away from zero. (See tests C46012A..Z (26 tests).)

3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

This implementation evaluates the 'LENGTH of each constrained array subtype during elaboration of the type declaration. This causes the declaration of a constrained array subtype with more than INTEGER'LAST (which is equal to SYSTEM.MAX_INT for this implementation) components to raise CONSTRAINT_ERROR. However the optimisation mechanism of this implementation suppresses the evaluation of 'LENGTH if no object of the array type is declared depending on whether the bounds of the array are static, the visibility of the array type, and the presence of local subprograms. These general remarks apply to points (1) to (5), and (8).

1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception if the bounds of the array are static. (See test C36003A.)

2) CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components if the bounds of the array are not static and if the subprogram declaring the array type contains no local subprograms. (See test C36202A.)

3) CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with SYSTEAM.MAX_INT + 2 components if the bounds

of the array are not static and if the subprogram declaring the array type contains a local subprogram. (See test C36202B.)

4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises CONSTRAINT_ERROR when the array type is declared if the bounds of the array are not static and if there are objects of the array type. (See test C52103X.)

5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the array type is declared if the bounds of the array are not static and if there are objects of the array type. (See test C52104Y.)

6) In assigning one-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

7) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

8) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises CONSTRAINT_ERROR when the array type is declared if the bounds of the array are not static and if there are objects of the array type. (See test E52103Y.)

f. Discriminated types.

1) In assigning record types with discriminants, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

1) In the evaluation of a multi-dimensional aggregate, the test results indicate that all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

2) In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

1) The pragma INLINE is supported for functions and procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

2) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

3) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

4) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)

6) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

7) Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

8) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

9) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output.

1) The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

2) The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. However this implementation raises USE_ERROR upon creation of a file for unconstrained array types. (See tests AE2101H, EE2401D, and EE2401G.)

3) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E, CE2102N, and CE2102P.)

4) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102F, CE2102I..J (2 tests), CE2102R, CE2102T, and CE2102V.)

5) Modes IN_FILE, OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K (3 tests).)

6) RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

7) RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K AND CE2102Y.)

8) RESET and DELETE operations are supported for text files. (See tests CE3102F..G (2 tests), CE3104C, CE3110A, and CE3114A.)

9) Overwriting to a sequential file truncates the file to the last element written. (See test CE2208B.)

10) Temporary sequential files are given names and deleted when closed. (See test CE2108A.)

11) Temporary direct files are given names and deleted when closed. (See test CE2108C.)

12) Temporary text files are given names and deleted when closed. (See test CE3112A.)

13) More than one internal file can be associated with each external permanent (not temporary) file for sequential files when reading only. (See tests CE2107B..E (5 tests), CE2107L, CE2110B, and CE2111D.)

14) More than one internal file can be associated with each external permanent (not temporary) file for direct files when reading only. (See tests CE2107G..H (3 tests), CE2110D and CE2111H.)

15) More than one internal file can be associated with each external permanent (not temporary) file for text files when reading only. (See tests CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 317 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 14 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	129	1131	2006	17	27	46	3356
Inapplicable	0	7	309	0	1	0	317
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	198	575	542	245	172	99	160	331	137	36	252	325	284	3356
N/A	14	74	138	3	0	0	6	1	0	0	0	44	37	317
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84N	CD2A84M	CD50110	CD2B15C	CD7205C	CD2D11B
CD5007B	ED7004B	ED7005C	ED7005D	ED7006C	ED7006D
CD7105A	CD7203B	CD7204B	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 317 tests were inapplicable for the reasons indicated:

a. The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

b. C34007P and C34007S are expected to raise CONSTRAINT_ERROR. This implementation optimizes the code at compile time on lines 205 and 221 respectively, thus avoiding the operation which would raise CONSTRAINT_ERROR and so no exception is raised.

c. C41401A is expected to raise CONSTRAINT_ERROR for the evaluation of certain attributes, however this implementation derives the values from the subtypes of the prefix at compile time as allowed by 11.6 (7) LRM. Therefore elaboration of the prefix is not involved and CONSTRAINT_ERROR is not raised.

d. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45614C	C45631C
C45632C	B52004D	C55B07A	B55B09C	B86001W
CD7101F				

e. C45531M..P (4 tests) and C45532M..P (4 tests) are inapplicable because the value of SYSTEM.MAX_MANTISSA is less than 48.

f. C47004A is expected to raise CONSTRAINT_ERROR whilst evaluating the comparison on line 51, but this compiler evaluates the result without invoking the basic operation qualification (as allowed by 11.6 (7) LRM) which would raise CONSTRAINT_ERROR and so no exception is raised.

g. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. These tests recompile package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

h. B86001X, C45231D, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER or SHORT_INTEGER.

i. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

j. B86001Z, B86001U and C35702B are not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT or SHORT_FLOAT.

k. C96005B is not applicable because there are no values of type DURATION'BASE that are outside the range of DURATION.

l. CD1009C, CD2A41A, CD2A41B, CD2A41E and CD2A42A..J (10 tests) are not applicable because this implementation imposes restrictions on 'SIZE length clauses for floating point types.

m. CD2A61I and CD2A61J are not applicable because this implementation imposes restrictions on 'SIZE length clauses for array types.

n. CD2A71A..D (4 tests), CD2A72A..D (4 tests), CD2A74A..D (4 tests) and CD2A75A..D (4 tests) are not applicable because this implementation imposes restrictions on 'SIZE length clauses for record types.

o. CD2A84B..I (8 tests), CD2A84K and CD2A84L are not applicable because this implementation imposes restrictions on 'SIZE length clauses for access types.

p. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

q. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

r. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

s. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

t. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

u. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

v. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

w. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

x. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

y. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

z. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

aa. CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.

ab. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

ac. CE2105A is inapplicable because CREATE with IN_FILE mode is not supported by this implementation for SEQUENTIAL_IO.

ad. CE2105B is inapplicable because CREATE with IN_FILE mode is not supported by this implementation for DIRECT_IO.

ae. CE2107B..E (4 tests), CE2107L, CE2110B are not applicable because multiple internal files cannot be associated with the same external (not temporary) file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

af. CE2107G..H (2 tests), CE2110D, and CE2111H are not applicable because multiple internal files cannot be associated with the same external (not temporary) file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

ag. CE3102F is inapplicable because text file RESET is supported by this implementation.

ah. CE3102G is inapplicable because text file deletion of an external file is supported by this implementation.

ai. CE3102I is inapplicable because text file CREATE with OUT_FILE mode is supported by this implementation.

aj. CE3102J is inapplicable because text file OPEN with IN_FILE mode is supported by this implementation.

ak. CE3102K is inapplicable because text file OPEN with OUT_FILE mode is not supported by this implementation.

al. CE3109A is inapplicable because text file CREATE with IN_FILE mode is not supported by this implementation.

am. CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A are not applicable because multiple internal files cannot be associated with the same external file when one or more files is

writing for text files. The proper exception is raised when multiple access is attempted.

an. EE2401D contains instantiations of package DIRECT_IO with unconstrained array types. This implementation raises USE_ERROR upon creation of such a file.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 14 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A	B24009A	B29001A	B38003A	B38009A	B38009B
B51001A	B91001H	BA1101E	BC2001D	BC2001E	BC3204B
BC3205B	BC3205D				

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the SYSTEAM Ada Compiler S.7000/BS2000 Version 1.81 was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the SYSTEAM Ada Compiler S.7000/BS2000 Version 1.81 using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host and target computer:	Siemens 7.530-B
Host and target operating system:	BS2000, Version 7.5A
Compiler:	SYSTEAM Ada Compiler S.7000/BS2000 Version 1.81

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the SIEMENS computer.

After the test files were loaded to disk, the full set of tests was compiled, linked, and all executable tests were run on the Siemens 7.530-B. Results were transferred to a VAX 8530, via FITSIE, where they were evaluated and archived.

The compiler was tested using command scripts provided by SYSTEAM KG and reviewed by the validation team. Tests were compiled using the command

```
/CALL &VERSION..COMPILE, (<source>)
```

and linked with the command

```
/CALL $ADA.LINK, <main>, <main>.EXE, LIST=<main>
```

Chapter B tests were compiled with the full listing option. A full description of compiler and linker options is given in Apendix E.

Tests were compiled, linked, and executed (as appropriate) using a single computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at SYSTEAM KG, Karlsruhe and was completed on Thursday 31th August 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

SYSTEAM KG has submitted the following Declaration of Conformance concerning the SYSTEAM Ada Compiler S.7000/BS2000, Version 1.31

Declaration of Conformance

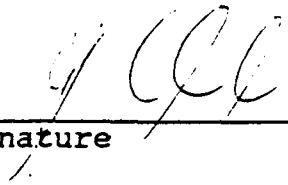
Customer:	SYSTEAM KG
Ada Validation Facility:	IABG m. b. H., Abt. SZT
ACVC Version:	1.10

Ada Implementation

Ada Compiler Name:	SYSTEAM Ada Compiler S.7000/BS2000
Version:	1.81
Host Computer System:	Siemens 7.530-B under BS2000, Version 7.5A
Target Computer System:	Siemens 7.530-B under BS2000, Version 7.5A

Customer's Declaration

I, the undersigned, representing SYSTEAM KG, declare that SYSTEAM KG has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation(s) listed in this declaration.



Signature

25.08.1989
Date

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependen. characteristics of the SYSTEAM Ada Compiler S.7000/BS2000 Version 1.81, as described in this Appendix, are provided by SYSTEAM KG. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

   ...

     type INTEGER is range - 2_147_483_648 .. 2_147_483_647;
     type SHORT_INTEGER is range - 32_768 .. 32_767;

     type FLOAT is digits 15 range
        - 16#0.FFFF_FFFF_FFFF_FF#E63 .. 16#0.FFFF_FFFF_FFFF_FF#E63;
     type SHORT_FLOAT is digits 6 range
        - 16#0.FFFF_FF#E63 .. 16#0.FFFF_FF#E63;

     type DURATION is delta 2#1.0#E-14 range
        - 131_072.0 .. 131_071.999_938_964_843_75;

   ...

end STANDARD;
```

7 Appendix F

This chapter, together with the Chapters 8 and 9, is the Appendix F required in [Ada], in which all implementation-dependent characteristics of an Ada implementation are described.

7.1 Implementation-Dependent Pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

7.1.1 Predefined Language Pragmas

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here. All the other pragmas listed in Appendix B of [Ada] are implemented and have the effect described there.

CONTROLLED
: has no effect.

INLINE
: Inline expansion of subprograms is supported with following restrictions:
the subprogram must not contain declarations of other subprograms, tasks, generic units or body stubs. If the subprogram is called recursively only the outer call of this subprogram will be expanded.

INTERFACE

is implemented for ASSEMBLER. The external subprograms written in assembly language must obey the calling conventions of the SYSTEAM Ada Compiler. An external subprogram is written like a normal assembly language procedure. The calling conventions are as follows:

On procedure entry

register	holds
R9	return address
R15	subprogram address
R8	parameter block address
R12	data area address; storage from 40(R12) upward can be used for local storage, 0(R12) to 39(R12) can be used for saving registers R4 to R13.

Registers

R9 to R13

must have identical contents on procedure entry and exit. It is strongly suggested to use the register save area indicated above applying appropriate STM and LM instructions; the 3 least significant bits of 39(R12) must be 0 to guarantee correct handling of errors in the subprogram.

The offsets of parameters within the parameter block are determined by the compiler. Array and record parameters are passed by reference. All other parameters are passed by value and result.

It is recommended to use only parameters of predefined types, access types and record types with a representation specification for the record layout.

The following table shows how the parameters have to be declared in the assembler source.

```
boolean          DS   X
character        DS   X
short_integer    DS   H
integer          DS   F
duration         DS   F
short_float      DS   E
float            DS   D

access type      DS   A
record type      DS   A
array type       DS   A
```

If each of the parameters occupies 4 bytes (i.e. is of type integer, duration, short_float, or of an access type, record type, or an array type) then the parameters are mapped in the order they occur in the subprogram declaration. If this type condition does not hold the implementation does not guarantee this order (in other words, the program is erroneous).

The following example shows the use of an external subprogram for accessing the BS2000 TMODE macro (the sequence number of the current task is output, package ebcdic is an implementation-defined library unit, see §5.3.3):

```
WITH ebcdic, text_io, system;
PROCEDURE external_example IS

SUBTYPE tskbyte IS integer RANGE 0 ..   16#FF#;
SUBTYPE tskhw   IS integer RANGE 0 .. 16#FFFF#;
TYPE tmode_data IS RECORD
   tsktype      : tskbyte;
   tskbufsz     : tskhw;
   tskpri       : tskbyte;
   tsktsn       : ebcdic.ebcdic_string (1..4);
   -- other components omitted
END RECORD;

PROCEDURE tmode (tm : IN system.address);
PRAGMA interface     (assembler , tmode);
PRAGMA external_name ("TMODE", tmode);

FOR tmode_data USE RECORD
   AT MOD 4;
   tsktype      AT  0 RANGE 0..7;
   tskbufsz     AT  1 RANGE 0..15;
   tskpri       AT  3 RANGE 0..7;
   tsktsn       AT  4 RANGE 0..31;
   -- other components omitted
END RECORD;
FOR tmode_data'size USE 48*8;

status : tmode_data;

BEGIN
   tmode (status'address);
   text_io.put_line (ebcdic.ascii_from_ebcdic(status.tsktsn));
END external_example;
```

The body of the external subprogram is written in assembly language as follows:

```
TMODE   CSECT
        STM 4,13,0(12)
        LR  2,15          TMODE macro changes R15
        USING TMODE,2
        USING PARAMS,8
        LA 1,40(12)
        USING DATA,1
*
        L   0,PARAMS
        ST  0,ADR
        LA  0,48
        STH 0,LEN
        TMODE     , TMODE assumes par.block address in R1
        LM 4,13,0(12)
        BR 9
*
DATA    DSECT       TMODE parameter block
ADR     DS 1F
LEN     DS 1H
PARAMS  DSECT       procedure parameter block
        DS 1A
*
        END
```

Assuming that the Ada procedure external_example is contained in the file EXAMPLE.ADA and that the body of the external subprogram is contained in the file EXAMPLE.ASM, the program can be compiled, linked and executed with the following commands:

```
/CALL $ADA.COMPILE,EXAMPLE.ADA

/SYSFILE SYSDTA=EXAMPLE.ASM
/EXEC ASSEMB
/CALL $ADA.SAVECODE,EXAMPLE.OBJ

/CALL $ADA.LINK,EXTERNAL_EXAMPLE,        -
/   EXAMPLE.EXE,EXTERNAL=EXAMPLE.OBJ

/EXEC EXAMPLE.EXE
```

MEMORY_SIZE
has no effect.

OPTIMIZE
has no effect.

PACK
see §8.1.

PRIORITY
There are two implementation-defined aspects of this pragma: First, the range of the subtype `priority`, and second, the effect on scheduling (§6) of not giving this pragma for a task or main program. The range of subtype `priority` is 0 .. 15, as declared in the predefined library package `system` (see §7.3); and the effect on scheduling of leaving the priority of a task or main program undefined by not giving pragma `priority` for it is the same as if the pragma `priority` 0 had been given (i.e. the task has the lowest priority).

SHARED
is supported.

STORAGE_UNIT
has no effect.

SUPPRESS
has no effect, but see §7.1.2 for the implementation-defined pragma `suppress_all`.

SYSTEM_NAME
has no effect.

7.1.2 Implementation-Defined Pragmas

EXTERNAL_NAME (<string>, <ada_name>)

<ada_name> specifies the name of a subprogram or of an object in a library package, <string> must be a string literal. It defines the external name of the specified entity. External names must not start with '#'; such strings are reserved for use by the SYSTEAM Ada System. The subprogram declaration of <ada_name> must precede this pragma. If several subprograms with the same name satisfy this requirement the pragma refers to that subprogram which precedes immediately. This pragma will be used in connection with pragma `interface (assembler)` (see §7.1.1).

RESIDENT (<ada_name>)

this pragma causes the value of the object <ada_name> to be held in storage (it may be held in a register too) and prevents assignments of a value to the object from being eliminated by the optimizer (see §3.2) of the SYSTEAM Ada Compiler. The following code sequence demonstrates the intended usage of the pragma:

```
...
x : integer;
a : SYSTEM.address;
   ...
BEGIN
  x := 5;
  a := x'ADDRESS;
  do_something (a);  -- let do_something be a non-local
                     -- procedure
                     -- a.ALL will be read in the body
                     -- of do_something
  x := 6;
  ...
```

If this code sequence is compiled by the SYSTEAM Ada Compiler with the option

```
OPTIMIZER=>ON
```

the statement `x := 5;` will be eliminated because from the point of view of the optimizer the value of x is not used before the next assignment to x. Therefore

```
PRAGMA resident (x);
```

should be inserted after the declaration of x.

This pragma can be applied to all those kinds of objects for which the address clause is supported (cf. §8.5).
It will often be used in connection with the pragma `interface (assembler, ...)` (see §7.1.1).

SQUEEZE
see §8.1.

SUPPRESS_ALL
causes all the run_time checks described in [Ada,§11.7] to be suppressed; this pragma is only allowed at the start of a compilation before the first compilation unit; it applies to the whole compilation.

7.2 Implementation-Dependent Attributes

The name, type and implementation-dependent aspects of every implementation-dependent attribute is stated in this chapter.

7.2.1 Language-Defined Attributes

The name and type of all the language-defined attributes are as given in [Ada]. We note here only the implementation-dependent aspects.

ADDRESS
The value delivered by this attribute applied to an object which occupies storage is the address of the storage unit where this object starts.
The value delivered by this attribute applied to a subprogram is the address of the storage unit where this subprogram starts.
For any other entity this attribute is not supported and will return the value `system.address_zero`.

IMAGE

The image of a character other than a graphic character (cf. [Ada, §3.5.5(11)]) is the string obtained by replacing each italic character in the indication of the character literal (given in [Ada, Annex C(13)]) by the corresponding upper-case character. For example, `character'image(`*nul*`) = "NUL"`.

MACHINE_OVERFLOWS

Yields `true` for each real type or subtype.

MACHINE_ROUNDS

Yields `false` for each real type or subtype.

STORAGE_SIZE

The value delivered by this attribute applied to an access type is as follows:
If a length specification (STORAGE_SIZE, see §8.2) has been given for that type (static collection), the attribute delivers that specified value.
In case of a dynamic collection, i.e. no length specification by STORAGE_SIZE given for the access type, the attribute delivers the number of storage units currently allocated for the collection. Note that dynamic collections are extended if needed.
If the collection manager (cf. §5.3.1) is used for a dynamic collection the attribute delivers the number of storage units currently allocated for the collection. Note that in this case the number of storage units currently allocated may be decreased by release operations.

The value delivered by this attribute applied to a task type or task object is as follows:
If a length specification (STORAGE_SIZE, see §8.2) has been given for the task type, the attribute delivers that specified value; otherwise, the default value is returned.

7.2.2 Implementation-Defined Attributes

There are no implementation-defined attributes.

7.3 Specification of the Package SYSTEM

The package system required in [Ada,§13.7] is reprinted here with all implementation-dependent characteristics and extensions filled in.

```
PACKAGE system IS

   TYPE designated_by_address IS LIMITED PRIVATE;
   TYPE address IS ACCESS designated_by_address;
   FOR address'size USE 32;
   FOR address'storage_size USE 0;

   address_zero : CONSTANT address := NULL;

   FUNCTION "+" (left : address; right : integer) RETURN address;
      PRAGMA _built_in (address_plus_integer, "+");

   FUNCTION "+" (left : integer; right : address) RETURN address;
      PRAGMA _built_in (integer_plus_address, "+");

   FUNCTION "-" (left : address; right : integer) RETURN address;
      PRAGMA _built_in (address_minus_integer, "-");

   FUNCTION "-" (left : address; right : address) RETURN integer;
      PRAGMA _built_in (address_minus_address, "-");


   SUBTYPE external_address IS STRING;

   -- External addresses use hexadecimal notation with characters
   -- '0'..'9', 'a'..'f' and 'A'..'F'. For instance:
   --    "7FFFFFFF"
   --    "80000000"
   --    "8" represents the same address as "00000008"

   FUNCTION convert_address (addr : external_address) RETURN address;
      -- convert_address raises CONSTRAINT_ERROR if
      -- the external address
      -- addr is the empty string, contains characters other than
      -- '0'..'9', 'a'..'f', 'A'..'F' or if the resulting address value
      -- cannot be represented with 32 bits.

   FUNCTION convert_address (addr : address) RETURN external_address;
      -- The resulting external address consists of
```

```
   -- exactly 8 characters  '0'..'9', 'A'..'F'.

TYPE name IS (s7000_bs2000);
system_name   : CONSTANT name := s7000_bs2000;

storage_unit : CONSTANT := 8;
memory_size  : CONSTANT := 2 ** 31;
min_int      : CONSTANT := - 2 ** 31;
max_int      : CONSTANT := 2 ** 31 - 1;
max_digits   : CONSTANT := 15;
max_mantissa : CONSTANT := 31;
fine_delta   : CONSTANT := 2.0 ** (-31);
tick         : CONSTANT := 1.0;

SUBTYPE priority IS integer RANGE 0 .. 15;

TYPE interrupt_number IS RANGE 1 .. 31;

FUNCTION interrupt_vector (number : interrupt_number) RETURN address;

PRAGMA inline (interrupt_vector);
-- converts an interrupt_number to an address;

TYPE interrupt_parameter IS RANGE 0 .. 16#FF_FFFF#;
FOR interrupt_parameter'size USE 4 * storage_unit;
-- Interrupt parameters are 24 Bit Values, Zero padded to
-- a 32 Bit word.

PROCEDURE trigger_interrupt (at_address : address;
                             parameter  : interrupt_parameter := 0);

PROCEDURE trigger_interrupt (at_address : address;
                             parameter  : address := address_zero);

non_ada_error : EXCEPTION;

-- non_ada_error is raised, if some event occurs which does not
-- correspond to any situation covered by Ada, e.g.:
--    illegal instruction encountered
--    error during address translation
--    illegal address


TYPE exception_id IS NEW integer;

no_exception_id     : CONSTANT exception_id := 0;
```

```
-- Coding of the predefined exceptions:

constraint_error_id : CONSTANT exception_id := ... ;
numeric_error_id    : CONSTANT exception_id := ... ;
program_error_id    : CONSTANT exception_id := ... ;
storage_error_id    : CONSTANT exception_id := ... ;
tasking_error_id    : CONSTANT exception_id := ... ;

non_ada_error_id    : CONSTANT exception_id := ... ;

status_error_id     : CONSTANT exception_id := ... ;
mode_error_id       : CONSTANT exception_id := ... ;
name_error_id       : CONSTANT exception_id := ... ;
use_error_id        : CONSTANT exception_id := ... ;
device_error_id     : CONSTANT exception_id := ... ;
end_error_id        : CONSTANT exception_id := ... ;
data_error_id       : CONSTANT exception_id := ... ;
layout_error_id     : CONSTANT exception_id := ... ;

time_error_id       : CONSTANT exception_id := ... ;


SUBTYPE return_code IS INTEGER;

success_code    : CONSTANT return_code := 0;
some_error_code : CONSTANT return_code := 4;
FUNCTION is_success (code : return_code) RETURN boolean;
     PRAGMA inline (is_success);
-- checks whether a return code indicates a success
-- is_success(success_code)    == true
-- is_success(some_error_code) == false

TYPE exception_information
    IS RECORD
       excp_id          : exception_id;
          -- Identification of the exception. The codings of
          -- the predefined exceptions are given above.
       code_addr        : address;
          -- Code address where the exception occured. Depending
          -- on the kind of the exception it may be be address of
          -- the instruction which caused the exception, or it
          -- may be the address of the instruction which would
          -- have been executed if the exception had not occured.
       error_code       : return_code;
   END RECORD;

PROCEDURE get_exception_information
```

```
            (excp_info : OUT exception_information);
      -- The subprogram get_exception_information must only be called
      -- from within an exception handler BEFORE ANY OTHER EXCEPTION
      -- IS RAISED. It then returns the information record about the
      -- exception handled currently.
      -- Otherwise, its result is undefined.

   PROCEDURE raise_exception_id
            (excp_id : exception_id);

   PROCEDURE raise_exception_info
            (excp_info : exception_information);

      -- The subprogram raise_exception_id raises the exception
      -- given as parameter. It corresponds to the RAISE statement.

      -- The subprogram raise_exception_info raises the exception
      -- described by the information record supplied as parameter.
      -- In addition to the subprogram raise_exception_id it allows to
      -- expicitly define all components of the exception
      -- information record.

      -- IT IS INTENDED THAT BOTH SUBPROGRAMS ARE USED ONLY WHEN
      -- INTERFACING WITH THE OPERATING SYSTEM.

   PROCEDURE system_call (cmd  : IN  string;
                          code : OUT return_code);

PRIVATE

   -- private declarations

END system;
```

7.4 Restrictions on Representation Clauses

See Chapter 8 of this manual.

7.5 Conventions for Implementation-Generated Names

There are implementation generated components but these have no names. (cf. §8.4 of this manual).

7.6 Expressions in Address Clauses

See §8.5 of this manual.

7.7 Restrictions on Unchecked Conversions

The implementation supports unchecked type conversions for all kind of source and target types with the restriction that the target type must not be an unconstrained array type. The result value of the unchecked conversion is unpredictable, if

```
target_type'SIZE > source_type'SIZE
```

7.8 Characteristics of the Input-Output Packages

The implementation-dependent characteristics of the input-output packages as defined in Chapter 14 of [Ada] are reported in Chapter 9 of this manual.

7.9 Requirements for a Main Program

A main program must be a parameterless library procedure. This procedure may be a generic instantiation; the generic procedure need not be a library unit.

7.10 Unchecked Storage Deallocation

The generic procedure unchecked_deallocation is provided, but the only effect of calling an instantiation of this procedure with an object X as actual parameter is

```
X := NULL;
```

i.e. no storage is reclaimed.

However, the implementation does provide an implementation-defined package collection_manager to support unchecked storage deallocation (cf. §5.3.1).

7.11 Machine Code Insertions

A package machine_code is not provided and machine code insertions are not supported.

7.12 Numeric Error

The predefined exception numeric_error is never raised implicitly by any predefined operation; instead the predefined exception constraint_error is raised.

8 Appendix F: Representation Clauses

In this chapter we follow the section numbering of Chapter 13 of [Ada] and provide notes for the use of the features described in each section.

8.1 Pragmas

PACK

As stipulated in [Ada,§13.1], this pragma may be given for a record or array type. It causes the Compiler to select a representation for this type such that gaps between the storage areas allocated to consecutive components are minimized. For components whose type is an array or record type the pragma pack has no effect on the mapping of the component type. For all other component types the Compiler will try to choose a more compact representation for the component type. All components of a packed data structure will start at storage unit boundaries and the size of the components will be a multiple of `system.storage_unit`. Thus, the pragma pack does not effect packing down to the bit level (for this see pragma **squeeze**).

SQUEEZE

This is an implementation-defined pragma which takes the same argument as the predefined language pragma pack and is allowed at the same positions. It causes the Compiler to select a representation for the argument type that needs minimal storage space (packing down to the bit level). For components whose type is an array or record type the pragma **squeeze** has no effect on the mapping of the component type. For all other component types the Compiler will try to choose a more compact representation for the component type. The components of a squeezed data structure will not in general start at storage unit boundaries.

8.2 Length Clauses

SIZE

for all integer, fixed point and enumeration types the value must be <= 32;
for `short_float` types the value must be = 32 (this is the amount of storage which is associated with these types anyway);
for `float` types the value must be = 64 (this is the amount of storage which is associated with these types anyway).
for access types the value must be = 32 (this is the amount of storage which is associated with these types anyway).
If any of the above restrictions are violated, the Compiler responds with a RESTRICTION error message in the Compiler listing.

STORAGE_SIZE

Collection size: If no length clause is given, the storage space needed to contain objects designated by values of the access type and by values of other types derived from it is extended dynamically at runtime as needed. If, on the other hand, a length clause is given, the number of storage units stipulated in the length clause is reserved, and no dynamic extension at runtime occurs.

Storage for tasks: The memory space reserved for a task is 10K bytes if no length clause is given (cf. Chapter 6). If the task is to be allotted either more or less space, a length clause must be given for its task type, and then all tasks of this type will be allotted the amount of space stipulated in the length clause (the activation of a small task requires about 1.4K bytes). Whether a length clause is given or not, the space allotted is not extended dynamically at runtime.

SMALL

there is no implementation-dependent restriction. Any specification for SMALL that is allowed by the LRM can be given. In particular those values for SMALL are also supported which are not a power of two.

8.3 Enumeration Representation Clauses

The integer codes specified for the enumeration type have to lie inside the range of the largest integer type which is supported; this is the type `integer` defined in package `standard`.

8.4 Record Representation Clauses

Record representation clauses are supported. The value of the expression given in an alignment clause must be 0, 1, 2, 4 or 8. If this restriction is violated, the Compiler responds with a RESTRICTION error message in the Compiler listing. If the value is 0 the objects of the corresponding record type will not be aligned, if it is 1, 2, 4 or 8 the starting address of an object will be a multiple of the specified alignment.

The number of bits specified by the range of a component clause must not be greater than the amount of storage occupied by this component. (Gaps between components can be forced by leaving some bits unused but not by specifying a bigger range than needed.) Violation of this restriction will produce a RESTRICTION error message.

There are implementation-dependent components of record types generated in the following cases :

- If the record type includes variant parts and if it has either more than one discriminant or else the only discriminant may hold more than 256 different values, the generated component holds the size of the record object.
- If the record type includes array or record components whose sizes depend on discriminants, the generated components hold the offsets of these record components (relative to the corresponding generated component) in the record object.

But there are no implementation-generated names (cf. [Ada,§13.4(8)]) denoting these components. So the mapping of these components cannot be influenced by a representation clause.

8.5 Address Clauses

Address clauses are supported for objects declared by an object declaration and for single task entries. If an address clause is given for a subprogram, package or task unit, the Compiler responds with a RESTRICTION error message in the Compiler listing.

If an address clause is given for an object, the storage occupied by the object starts at the given address. Address clauses for single entries are described in §8.5.1.

8.5.1 Interrupts

On SIEMENS/BS2000, all hardware interrupts are handled by BS2000. It is not possible to handle the hardware interrupts directly; however, some system services allow a process to be interrupted when a particular event occurs. The interrupt transfers control to a user-specified routine that handles the event.

BS2000 delivers an event when requested to do so, for instance by calls of UPAM, IPC, DCAM, TCS and CJC system services (cf. [BS2000, Makroaufrufe an den Ablaufteil, §3.3]). As parameter to a call of a system service the address of a user supplied event handler will usually be specified.

An address clause for an entry associates the entry with an interrupt at the SYSTEAM Ada System runtime system level, and the latter is associated with a BS2000 event by an appropriate user supplied BS2000 event handler. When an event occurs, the event handler initiates the entry call; the calling task and the called task continue their execution in parallel.

By this mechanism, an interrupt acts as an entry call to that task; such an entry is called an *interrupt entry*. An interrupt causes the ACCEPT statement corresponding to the entry to be executed.

The interrupt is mapped to an *ordinary* entry call. The entry may also be called by an Ada entry call statement. However, it is assumed that when an interrupt occurs there is no entry call waiting in the entry queue. Otherwise, the program is erroneous and behaves in the following way:

- If an entry call stemming from an interrupt is already queued, this previous entry call is lost.
- The entry call stemming from the interrupt is inserted into the front of the entry queue, so that it is handled before any entry call stemming from an Ada entry call statement.

8.5.1.1 Association between Entry and Interrupt

The association between an entry and an interrupt is achieved via an interrupt number (type `system.interrupt_number`), the range of interrupt numbers being 1 .. 31 (this means that 31 single entries can act as interrupt entries). A single entry of a task which has one `IN` parameter of type `integer` or `system.address` can be associated with an interrupt number by an address clause (the Compiler does not check these conventions). Since an address value must be given in the address clause, the interrupt number has to be converted into type `system.address`. The function `system.interrupt_vector` is provided for this purpose.

The following example associates the entry `intr_cmd` with the interrupt number 17.

```
...
TASK handler IS

   ENTRY intr_cmd (msg : IN ebcdic_string);

   FOR intr_cmd   USE AT system.interrupt_vector (17);
END;
...
```

The task body contains an ordinary accept statement for the entry.

8.5.1.2 Association between Interrupt and Event Handlers

When an entry is to be called via a BS2000 event, the user supplies an event handler for this BS2000 event using the appropriate BS2000 macro (e.g. STXIT); the event handler must give control to the SYSTEAM Ada System by the POSSIG macro stating event id #ADAINTR and using the 4 byte Post Code value of POSSIG as follows (cf. [BS2000, Makroaufrufe an den Ablaufteil, §3.3.5]):

- byte 0 contains the interrupt number,
- bytes 1 to 3 can contain the IN parameter of the interrupt entry.

The effect of the execution of the event handler given by the user depends on whether there is a task currently waiting at an ACCEPT or selective wait statement for an entry which is associated with the interrupt number given in the POSSIG Post Code.

If there is a task waiting, a rendezvous with that task is performed immediately. If several tasks are waiting for the same interrupt, the program is erroneous (cf. [Ada, §13.5(8)]) and a rendezvous is performed with any of these tasks.

Otherwise, the information that the interrupt nr occurred, and the corresponding parameter value, are stored by the Ada runtime system. If later on a task performs an ACCEPT or selective wait statement for the entry associated with the interrupt nr the rendezvous is performed.

Therefore an interrupt is never treated as a conditional entry call. If the interrupt nr occurs again before the previous one has been handled, the previous one is lost.

The following example shows an event handler for the BS2000 /INTR command, which connects to the entry intr_cmd given above; the user program can then examine the message text given with the /INTR command:

```
INIT     CSECT
         USING *,15
         STXIT TYP=CO,INTR=(HANDLER,127),INTRBUF=INTRBUF
         BR 9
*
HANDLER  BALR 15,0
         USING *,15
         EXTRN #ADAINTR
         LA 2,INTRBUF                      * entry parameter
         ICM 2,X'8',=F'17'                 * interrupt number 17
         POSSIG EIID=#ADAINTR,SPOSTR=2     * Post Code in R2
*
INTRBUF  DS OF
         DS CL64
         END
```

8.6 Change of Representation

The implementation places no additional restrictions on changes of representation.

9 Appendix F: Input-Output

In this chapter we follow the section numbering of Chapter 14 of [Ada] and provide notes for the use of the features described in each section.

9.1 External Files and File Objects

The total number of open files (excluding the two standard files) must not exceed 14. Any attempt to exceed this limit raises the exception `use_error`.

The only form of file sharing which is allowed is shared reading. If two or more files are associated with the same external file at one time (regardless of whether these files are declared in the same program or task), all of these (internal) files must be opened with the mode `in_file`. An attempt to open one of these files with a mode other than `in_file` will raise the exception `use_error`.

The following restrictions apply to the generic actual parameter for `element_type`:

- input/output of access types is not defined.
- input/output of unconstrained array types is only possible with a variable record format.
- the size of an object to be input or output must not be greater than 2048 storage units; there is one exception for sequential files connected to BS2000 SAM files, see §9.2.1.2. Further restrictions depending on BS2000 data access methods are stated below.
- input/output is not possible for an object whose (sub)type has a size which is not a multiple of `system.storage_unit`. Such objects can only exist for types for which a representation clause or the pragma `squeeze` is given. `Use_error` will be raised by any attempt to read or write such an object or to open or create a file for such a (sub)type.

9.2 Sequential and Direct Files

Sequential and direct files are represented by BS2000 SAM, ISAM, PAM or EAM files with fixed-length or variable-length records. Each element of the file is stored in one record.

9.2.1 File Management

Since there is a lot to say about this section, we shall introduce subsection numbers which do not exist in [Ada].

9.2.1.1 The NAME and FORM Parameters

The NAME parameter must be empty (for temporary files), a BS2000 file specification or a BS2000 link name. The FORM parameter selects whether the string is regarded as a file specification or as a link name (see below).

As file specification any complete, fully qualified BS2000 file name is allowed. The general form is:

[:<catid>:] [$*userid.*] <filename>

The <catid> and/or the *userid* may be omitted, in which cases the defaults of the operating system are used.

The total length of the file specification must not exceed 54 characters. All letters contained in the name string must be in upper case.

The function NAME will return the file specification of the external file associated with the internal file. Any normalizations performed by BS2000 on the supplied name string during the create/open-operation are transparent. This also means that in the case where the name string is really a linkname, the function NAME does not return the supplied linkname but the name of the associated file. For temporary files, i.e. if the name string is empty, #A.<TSN>.<nn> is chosen as file name where <TSN> is the 4 digit Task Serial Number and <nn> is a 2 digit decimal number. Note that the #-character is expanded during the open/create-operation if the BS2000 installation supports temporary files.

The exception NAME_ERROR is raised if the name string does not represent a legal BS2000 file specification string; for instance, if it contains illegal characters like lower case letters, is too long or syntactically incorrect. Note that the file specification string must not contain wild cards even if a unique file is specified.

The FORM string may contain from 0 to 3 upper case characters; the empty string causes the default form settings to be taken. Legal characters may occur in any order

but only once. Blanks are ignored but contribute to the length of the string. Possible characters for the form string and their meanings are listed below.

The characters selecting the BS2000 file access method are:

S	sequential access method (SAM)
I	index sequential access method (ISAM)
P	primary access method (PAM)
E	evanescent access method (EAM)

The characters selecting the BS2000 record format are:

V	variable record format
F	fixed record format

The character "=" in the form string causes the name string to be treated as a BS2000 linkname. In this case, a file must be associated with that linkname (e.g. using the BS2000-command: /FILE ...,LINK=linkname) before the create/open-operation is executed. If the "=" character is missing the name string is treated as a file name.

If an already existing file is opened the access method and the record format need not be supplied in the form string. If they are omitted, the corresponding information is retrieved from the file catalogue of the operating system. However, if the access method and/or record format characters are supplied in the form string but are not consistent with the file catalogue the exception USE_ERROR is raised during OPEN.

If a new file is created and the access method and/or record format characters are not present in the form string, defaults are used.

The legal form specifications depend on which predefined package is used; they are listed in the table below; optional characters are enclosed in brackets []:

Ada package used	legal form specifications	default create	default open
`sequential_io`	`[S][V][=]`	SV	SV

`direct_io`	`[I][V][=]`
	`P[F][=]`
	`E[F]`
		IV	from catalogue

`text_io`	`[S][V][=]` `I[V][=]`	SV	from catalogue

Examples of legal form strings :

```
"" "S" "S=" "=S" " S " "SV=" "I" "P" "EF"
```

Examples of illegal form strings :

"SI"	ambiguous access method
" IV="	more than 3 characters long
"IF"	fixed record format not allowed for ISAM files

The function FORM returns the form string that was supplied to the create open operation without any modification.

9.2.1.2 Sequential Files

A sequential file is represented by a BS2000 sequential file with variable-length records.

By default the object size must not exceed 2044 bytes. However due to the use of BS2000 SAM Locating Mode, objects up to a size of 32_764 bytes (BS2000 DMS restriction) can be read and written when the file is associated by linkname (FORM parameter value "=") and the file attributes are set appropriately. For example, the BS2000 JCL command

```
/FILE externalname,LINK=linkname,FCBTYPE=SAM,RECFORM=V,    -
/                  BLKSIZE=(STD,12),SPACE=(12,12)
```

can be used to connect to a file of objects up to a size of 12*2048-4 bytes.

9.2.1.3 Direct Files

The implementation dependent type COUNT defined in the package specification of DIRECT_IO has an upper bound of :

COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)

Direct files are represented by BS2000 indexed files with variable-length records or by PAM or EAM files with fixed length records.

For indexed files the record index is stored as an unsigned binary value in the first four bytes of each record (implying a key length of 4); the maximum record size is set to ELEMENT_TYPE'SIZE / SYSTEM.STORAGE_UNIT by default. The object size must not exceed 2040 bytes.

For PAM or EAM files the object size must not exceed 2048 bytes. Each page holds a single object.

9.3 Text Input-Output

Text on external BS2000 files is coded in (8 bit) EBCDIC, while the internal representation within Ada programs is in ASCII code. When text files are read or written their contents are transformed from EBCDIC to ASCII and vice versa by special I/O functions. This also affects the line/page/file terminators.

The underlying I/O implementation generates (and expects) these terminators according to the following rules.

Text files are represented as SAM or ISAM files with variable record format. Each line is represented as a sequence of one or more records; all records except the last one contain a continuation marker (EBCDIC 16#2F#) as the last character. This character does not belong to the line. Lines of length less than 256 characters are represented as single records and, hence, do not contain continuation markers.

Line terminators are not represented explicitly unless followed by a page terminator. The only exception is the empty line, which is represented as a single record containing EBCDIC 16#0D# (=<CR>) as its only character. A line terminator followed by a page terminator is represented by EBCDIC 16#0C# (=<FF>) behind the last character of the last line of a page, i.e. the last character of the last record which belongs to the last line of the page is EBCDIC 16#0C#. A line terminator followed by a page terminator followed by a file terminator is represented in the external file by EBCDIC 16#03# (=<ETX>).

Text files established under BS2000 which do not contain any of the above format effectors may be read by Ada programs without difficulty. However, these files will be considered as consisting of one page only. End of file is recognized correctly whether or not the line-page-file-terminator EBCDIC 16#03# is present.

When processing text files written by Ada programs it has to be kept in mind that these files may contain format effectors. Even if page terminators and empty lines are avoided there will always be a line-page-file terminator EBCDIC 16#03# at the end of the file.

Of course, format effectors may be inserted into existing text files with the help of a system editor.

9.3.1 File Management

The meaning of the FORM parameter for text files is defined in §9.2.1.1.

9.3.2 Default Input and Output Files

The standard input and output file are associated with SYSDTA and SYSOUT, respectively. If a program reads from the standard input file or writes to the standard output file the SYSFILE command may be used as usual to redirect SYSDTA or SYSOUT.

9.3.3 Implementation-Defined Types

The implementation dependent types COUNT and FIELD defined in the package specification of TEXT_IO have the following upper bounds :

COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)

FIELD'LAST = 255

9.4 Exceptions in Input-Output

For each of `name_error`, `use_error`, `device_error` and `data_error` we list the conditions under which that exception can be raised. The conditions under which the other exceptions declared in the package `io_exceptions` can be raised are as described in [Ada,§14.4].

NAME_ERROR

- in an open operation, if the specified file does not exist;
- if the `name` parameter in a call of the `create` or `open` procedure is not a legal BS2000 file specification string; for example, if it contains illegal characters, is too long or is syntactically incorrect; and also if it contains wild cards, even if that would specify a unique file.

USE_ERROR

- if an attempt is made to increase the total number of open files (excluding the two standard files) to more than 14;
- whenever an error occurred during an operation of the underlying BS2000 system. This may happen if an internal error was detected, an operation is not possible for reasons depending on the file or device characteristics, a size restriction is violated, a capacity limit is exceeded or for similar reasons;
- if the characteristics of the external file are not appropriate for the file type; for example, if the record size of a file with fixed-length records does not correspond to the size of the element type of a `direct_io` or `sequential_io` file. In general it is only guaranteed that a file which is created by an Ada program may be reopened by another program if the file types and the form strings are the same;
- if two or more (internal) files are associated with the same external file at one time (regardless of whether these files are declared in the same program or task), and an attempt is made to open one of these files with mode other than `in_file`. However, files associated with terminal devices (which is only legal for text files) are excepted from this restriction. Such files may be opened with an arbitrary mode at the same time and associated with the same terminal device;
- if a given `form` parameter string does not have the correct syntax or if a condition on an individual form specification described in §9.2.1.1 is not fulfilled;
- if an attempt is made to open or create a sequential or direct file for an element type whose size is not a multiple of `system.storage_unit`; or if an attempt is made to read or write an object whose (sub)type has a size which is not a multiple of `system.storage_unit` (such situations can only arise for types for which a representation clause or the pragma `squeeze` is given);

DEVICE_ERROR
is never raised. Instead of this exception the exception use_error is raised whenever an error occurred during an operation of the underlying BS2000 system.

DATA_ERROR The conditions under which data_error is raised by text_io are laid down in [Ada]; the following notes apply to the packages sequential_io and direct_io:

- by the procedure read if the size of a variable-length record in the external file to be read exceeds the storage size of the given variable or else the size of a fixed-length record in the external file to be read exceeds the storage size of the given variable which has exactly the size element_type'SIZE.
- by the procedure read if an element with the specified position in a direct file does not exist; this is only possible if the file is associated with a relative or an indexed file.
- In general, the exception data_error may not necessarily be raised by the procedure read if the element read is not a legal value of the element type.

9.5 Low Level Input-Output

We give here the specification of the package low_level_io:

```
PACKAGE low_level_io IS

   TYPE device_type IS (null_device);

   TYPE data_type IS
      RECORD
         NULL;
      END RECORD;

   PROCEDURE send_control    (device : device_type;
                              data   : IN OUT data_type);

   PROCEDURE receive_control (device : device_type;
                              data   : IN OUT data_type);

END low_level_io;
```

Note that the enumeration type `device_type` has only one enumeration value, `null_device`; thus the procedures `send_control` and `receive_control` can be called, but `send_control` will have no effect on any physical device and the value of the actual parameter `data` after a call of `receive_control` will have no physical significance.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below:

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	254 * 'A' & '1'
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	254 * 'A' & '2'
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	127 * 'A' & '3' & 127 * 'A'

Name and Meaning	Value
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	127 * 'A' & '4' & 127 * 'A'
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	252 * '0' & "298"
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	250 * '0' & "690.0"
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	'"' & 127 * 'A' & '"'
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	'"' & 127 * 'A' & '1' & '"'
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	235 * ' '
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	2_147_483_648
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	s7000_bs2000
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	255
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_FIXED_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_FLOAT_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	0.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	200_000.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	15
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	invalid_1.!@#$%^&*()
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	invalid_2.!@#$%^&*()

Name and Meaning	Value
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-0.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-200_000.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	255
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_648

Name and Meaning	Value
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"2:" & 250 * '0' & "11:"
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"16:" & 248 * '0' & "F.E:"
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	'"' & 253 * 'A' & '"'
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_TYPE
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	s7000_bs2000
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#

Name and Meaning	Value
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	2_147_483_648
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	s7000_bs2000
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	1.0

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C This test expects that the string "-- TOP OF PAGE. -- 63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this that must appear at the top of the page.

b. A39005G This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

c. B97102E This test contains an unitended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

d. C97116A This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING-_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

e. BC3009B This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

f. CD2A62D This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests] These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them

to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

h. CD2A81G, CD2A83G, CD2A84N & M, & CD50110 [5 tests] These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86 & 96, 86 & 96, and 58, resp.).

i. CD2B15C & CD7205C These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

j. CD2D11B This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

k. CD5007B This test wrongly expects an implicitly declared subprogram to be at the the address that is specified for an unrelated subprogram (line 303).

l. ED7004B, ED7005C & D, ED7006C & D [5 tests] These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

m. CD7105A This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

n. CD7203B, & CD7204B These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

o. CD7205D This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

p. CE2107I This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective

is not considered valid. (line 90)

q. CE3111C This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A This test contains several calls to END_OF_LINE & END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, & 136).

s. CE3411B This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER AND LINKER OPTIONS

This appendix contains information concerning the compilation and linkage commands used within the command scripts for this validation.

3 Compiling, Linking and Executing a Program

3.1 Overview

After a program library has been created, one or more compilation units can be compiled in the context of this library. The compilation units can be placed on different source files or they can all be on the same file. One unit, a parameterless procedure, acts as main program. If all units needed by the main program and the main program itself have been compiled successfully, they can be linked. The resulting code can then be executed by giving an EXEC command.

§3.2 and §3.4 describe in detail how to call the Compiler and the Linker. Further on in §3.3 the Completer, which is called to generate code for instances of generic units and to complete packages in the program which do not require a body, is described.
§3.5 explains the information which is given if the execution of a program is abandoned due to an unhandled exception.
The information the Compiler produces and outputs in the Compiler listing is explained in §3.6.
Finally, the log of a sample session is given in §3.7.

3.2 Starting the Compiler

To start the SYSTEAM Ada Compiler, call the command

```
/CALL $ADA.COMPILE, <source>  [, LIBRARY=<library>]
                              [, OPTIONS=<string> ]
                              [, LIST=<filename>  ]
```

The input file for the Compiler is <source>. The maximum length of lines in <source> is 255; longer lines are cut and an error is reported.

<library> is the name of the program library; ADALIB is assumed if this parameter is not specified. The library must exist (see §2.2 for information on program library management).

The compiler listing is written to the file selected by <filename>.LIS if the LIST-parameter is present. Otherwise

```
<source>.LIS
```

is chosen as the name of the listing file, where a leading *$userid* is removed from <source> if present. The resulting length of the listing file name must not exceed the system limits. BS2000 Version 7.5 supports 41 characters.

Options for the Compiler can be specified by using the parameter OPTIONS; they have an effect only for the current compilation. <string> must have the syntax

```
'[option {, option}]'
```

where blanks are allowed following and preceding lexical elements within the string; quotes can be omitted around a single option.

The Compiler accepts the following options:

LIST => ON/OFF	(default is OFF)
OPTIMIZER => ON/OFF	(default is ON)
INLINE => ON/OFF	(default is ON)
COPY_SOURCE => ON/OFF	(default is OFF)
SUPPRESS_ALL	
SYMBOLIC_CODE	

The options LIST and SUPPRESS_ALL have the same effect as the corresponding pragmas would have at the beginning of the source (see [Ada,Appendix B] and §7.1.2 of this manual).

No optimizations like constant folding, dead code elimination or common subexpression elimination are done if OPTIMIZER => OFF is specified.

Inline expansion of subprograms which are specified by a pragma `inline` (cf. §7.1.1) in the Ada source can be suppressed generally by giving the option INLINE => OFF. The value ON will cause inline expansion of the respective subprograms.

COPY_SOURCE => ON causes the Compiler to copy the source file <source> into the program library. After compilation with COPY_SOURCE => OFF the program source (without comments) can still be generated using the SYSTEAM Ada System tool SOURCEGEN (cf. [ST21/84]).

A symbolic code listing can be produced by specifying the option SYMBOLIC_CODE when calling the Compiler. The code listing is written on a file `<filename>.SYM` or `<source>.SYM`, depending on the LIST parameter, as for the listing file.

The source file may contain a sequence of compilation units, cf. §10.1 of [Ada]. All compilation units in the source file are compiled individually. When a compilation unit is compiled successfully, the program library is updated and the Compiler continues with the compilation of the next unit on the source file. If the compilation unit contained errors, they are reported (see §3.6). In this case, no update operation is performed on the program library and all subsequent compilation units in the compilation are only analyzed without generating code.

The COMPILE procedure leaves process switch 1 set (ON) if one of the compilation units contained errors.

3.3 The Completer

The Compiler does not generate code for instances of generic bodies. Since this must be done before a program using such instances can be executed, the COMPLETER tool must be used to complete such units. The COMPLETER must also be called to complete packages in the program which do not require a body. These things are done implicitly when LINK is called (with parameter COMPLETE=ON).

It is also possible to call the Completer explicitly by

```
/CALL $ADA.COMPLETE, <ada_name>
                     [, LIBRARY=<library>]
                     [, OPTIONS=<string> ]
                     [, LIST=<filename> ]
```

<ada_name> must be the name of a library unit. All library units that are needed by that unit (cf. [Ada,§10.5]) are completed, if possible, and so are their subunits, the subunits of those subunits and so on. The meaning of the parameters LIBRARY and LIST corresponds to that of the COMPILE command (cf. §3.2). Options apply to all units that are completed; the following ones are accepted (cf. §3.2):

OPTIMIZER => ON/OFF
INLINE => ON/OFF
SUPPRESS_ALL
SYMBOLIC_CODE

The COMPLETE procedure leaves process switch 1 set (ON) if it detected any errors.

In this case a listing file containing the error messages (cf. §3.6) is created. When no LIST parameter is given, the listing file name will be COMPLETE.LIS, otherwise it will be <filename>.LIS.

3.4 The Linker

An Ada program is a collection of units used by a main program which controls the execution. The main program must be a parameterless library procedure; any parameterless library procedure within a program library can be used as a main program.

The Siemens/BS2000 system linker TSOSLNK is used by the SYSTEAM Ada System Linker.

To link a program, call the command

```
/CALL $ADA.LINK, <ada_name>
                 , <filename>
                [, LIBRARY=<library>       ]
                [, COMPLETE=ON/OFF         ]
                [, OPTIONS=<string>        ]
                [, LIST=<filename>         ]
                [, IDA=YES/NO              ]
                [, LINKOPT=<params>        ]
                [, MAP=<mapfilename>       ]
                [, SELECT=ON/OFF           ]
                [, EXTERNAL=<extcodefile> ]
                [, MODLIB=NO/YES/SHARE     ]
                [, CODELABEL=<codelabel>   ]
                [, CODEENTRY=<codeentry>   ]
                [, STARTER=<starterfile>   ]
```

<ada_name> is the name of the library procedure which acts as the main program.

<filename> is the name of the file which is to contain the executable code after linking; this is either a load module file or an LMR library, depending on the MODLIB parameter.

<library> is the name of the program library which contains the main program; ADALIB is assumed if this parameter is not specified.

The COMPLETE parameter specifies whether the program is to be completed before it is linked; default is ON. If the Completer is called, the parameters LIBRARY and OPTIONS are passed to it (cf. §3.3).

The following options are relevant for the Pre-Linker (cf. §3.2):

OPTIMIZER => ON/OFF
SYMBOLIC_CODE

The listing (symbolic code listing) is written to file LINK.LIS (LINK.SYM) or <filename>.LIS (<filename>.SYM) depending on the presence of the LIST parameter.

The IDA parameter specifies whether ISD information for the Siemens BS2000 Debugger IDA is to be generated; default is YES.

The LINKOPT parameter can be used to pass parameters directly to TSOSLNK when MODLIB=NO, <params> must be quoted and must begin with a ",";
example: LINKOPT=',SORT=Y,XREF=Y'.

If the MAP parameter is given, the map listing of the BS2000 Linker TSOSLNK is preserved in the specified file. If IDA=YES, a map with all symbols is given; otherwise the map contains only minimal information.

SELECT=ON causes the object code of subprogram bodies to be included in the executable program only if this subprogram may be called during program execution. In the case of OFF the code of all compilation units mentioned in a context clause (in a transitive manner) is linked together; the default is ON.

The EXTERNAL parameter specifies a list of object files separated by | which contain object modules of those program units which are not written in Ada (e.g. object modules of subprograms written in assembly language). For those program units the pragmas

```
PRAGMA interface (assembler, ... )     -- (cf. §7.1.1)
```

and

```
PRAGMA external_name ( ... )     -- (cf. §7.1.1)
```

must be given in the Ada source.

An object file is generated in the necessary format by assembling resp. compiling the external programs into the *-file and then copying the *-file into the object file:

```
/CALL $ADA.SAVECODE ,<objectfile>
```

The LINK command supports three kinds of result files as selected by the MODLIB parameter, the default is MODLIB=NO:

- MODLIB=NO: The LINK command produces a load module (to be executed via /EXEC <filename>). The main linker generates two object modules, one containing the code (write protected, if linked with IDA=NO, otherwise not write protected), one containing the data, i.e. statically allocated variables (not write protected). Both object modules request alignment on page boundaries. The modules are stored on the *-file. The system linker TSOSLNK generates the load module and stores it on the result file.

- MODLIB=YES: The LINK command produces a LMR library containing two object modules, the code module and the data module. The Ada program can be started using the dynamic loader and linker DLL

  /EXEC (#PROGRAM,<filename>).

  Also, a load module may be generated using the TSOSLNK.
- MODLIB=SHARE: The LINK command produces an LMR library <filename>, which contains a shareable code module and a data module, and a second file <starterfile>, specified by parameter STARTER, containing the starter program for executing the program when its code module is loaded in shared code. This is of interest in case of larger Ada programs: The space otherwise occupied by the code may then be used as data space. The code module defines two labels, which have to be provided by the user through parameters CODELABEL and CODEENTRY, and which do not conflict with other labels in the name space of shared code.

The parameters CODELABEL, CODEENTRY and STARTER are only required when MODLIB=SHARE.

The following steps are performed during linking. First the Completer is called, unless suppressed by COMPLETE=OFF, to complete the bodies of instances and to complete packages in the program which do not require a body. Then the Pre-Linker is executed; it determines the compilation units that have to be linked together and a valid elaboration order. A code sequence to perform the elaboration is generated. Then the Main Linker is used to link the modules of the Ada program, the runtime system and external modules (if requested by parameter EXTERNAL), yielding a data and a code object module. Finally BS2000 TSOSLNK is used to link these modules statically, producing a BS2000 load module (MODLIB=NO), or LMR is used to generate a library containing the two modules. For MODLIB=SHARE the BS2000 ASSEMB and TSOSLNK are used to assemble and link the starter program.

The LINK procedure leaves process switch 1 set (ON) if one of the above mentioned steps failed (e.g. if one of the completed units contained errors, if any compilation unit could not be found in the program library or if no valid elaboration order could be determined because of incorrect usage of the pragma elaborate).

3.5 Executing a Program

After linking, the program can be executed as follows, depending on the use of the MODLIB parameter at link time:

For MODLIB=NO the program is executed by

```
/EXEC <filename>
```

For MODLIB=YES the program is executed by

```
/EXEC (#PROGRAM,<filename>)
```

For MODLIB=SHARE the code module of the program has to be loaded into shared code by the system administrator by

```
/SHARE (<codelabel>),<filename>
/LOAD (<codelabel>,<filename>)
```

and is then executed by

```
/SYSFILE TASKLIB=<filename>
/EXEC <starterfile>
```

If an Ada program is abandoned due to an unhandled exception, a message is displayed as in the following example:

```
%  P500 ADAPGM/000/89-03-11 LOADED
*** Ada program abandoned due to unhandled exception!
    exception    : CONSTRAINT_ERROR
    raised at    : 5C0010AC
```